Message efficient Byzantine Reliable Broadcast protocols on known topologies

Tim Anema[1] , **Jérémie Decouchant**[1]
[1]TU Delft

Abstract

In this paper, we consider the Reliable Communication and Byzantine Reliable Broadcast problems on partially connected networks with authenticated links. We consider the Reliable Communication (RC) problem on partially connected networks, and the Byzantine Reliable Broadcast (BRB) problem on partially and fully connected networks. Danny Dolev's protocol works on the former, while Gabriel Bracha's *authenticated double echo* protocol works on the latter in the case of a fully connected network. By layering the two protocols the BRB problem can be solved for partially connected networks. The state-of-the-art protocols for these problems focus on unknown topologies, whereas we focus on known topologies. We show that these protocols can be optimized when processes leverage this knowledge. Our simulations with our profiler show that we can drastically reduce the message complexity and network usage (e.g., a reduction of 71.9% and 79.4% respectively with a 12B payload when N=150 and f=20 for Dolev) compared to naive routing with our optimizations and disjoint path solver.

1 Introduction

Distributed systems are at the heart of our everyday lives. These systems consist of autonomous processes that communicate with a subset of other processes to coordinate their efforts. This means that these systems have to be robust against arbitrary behavior that some faulty or malicious nodes might exhibit. Fault-tolerant distributed communication algorithms are being used in practice to give this guarantee.

The Byzantine fault model is often used to describe these fault-tolerant systems. In this model there are two types of processes: correct processes which follow their programming faithfully, and Byzantine processes that exhibit arbitrary behavior which includes but is not limited to altering messages, creating new ones, or dropping messages altogether.

There are several solutions to this problem, all of which make different assumptions and differ in their guaranteed properties. An example of this is Dolev's *reliable communication* (RC) algorithm [1], which assumes a $2f + 1$-connected network. Another example is Bracha's double echo authenticated broadcast [2], which assumes a fully connected network. The state-of-the-art solution for *Byzantine Reliable Broadcasts* (BRB) described by Wang and Wattenhofer [3] and improved by Bonomi et al. [4] relies on an optimized combination of Dolev's RC algorithm [5] and Bracha's double echo authenticated broadcast.

This research will focus on optimizing Dolev, Bracha, and Bracha-Dolev by minimizing the number of redundant messages transmitted when the topology of the network is known to all processes. While the problem of reducing the amount of messages has been discussed in several papers, they focus on unknown network topologies [4; 5; 6], introduce cryptography and/or *public-key infrastructure* (PKI) [7; 8], or use trusted nodes [9]. Focusing on the case where the network topology is known to all processes is worth investigating, as this is a realistic use case and might allow for more optimizations. In addition to this, other papers have shown ways to reconstruct the topology [10] with protocols such as `Explorer` [11] and `Explorer2` [12], which makes it possible to use the optimizations in this paper for previously unknown topologies. Even though the aforementioned papers do not assume a known network, most of their optimizations also apply.

In this paper, we will start from solutions using naive routing and make the following contributions:
(i) We explain how a routing table can be created for Dolev using a combination of existing algorithms.
(ii) We discuss how the verification step of Dolev is trivial in our system.
(iii) We introduce 9 modifications to Dolev, Bracha, and Bracha-Dolev.
(iv) We present a detailed performance analysis using our profiling tool.

The structure of this paper is as follows. We will first explain what work has already been done in this field. Sec. 2 will introduce the system model and the problem, while Sec. 3 will provide some background on Dolev, Bracha, and Bracha-Dolev. Sec. 4, 5 and 6 will then introduce our novel modifications for Dolev, Bracha and Bracha-Dolev respectively. Sec. 7 contains our performance analysis. We will briefly discuss the broader impact and reproducibility of our work in Sec. 8. Finally, Sec. 9 will conclude our paper.

Related work

The idea of reliably reaching an agreement in the presence of faulty or malicious processes was first introduced by Lamport et al. [13] and was named the *Byzantine Agreement*. The network tolerance to faults can be represented as f, which represents the number of Byzantine processes that can be present before correct processes can no longer reliably communicate with each other. One can imagine this number heavily depends on the network's connectivity, i.e., the number of nodes that can fail before the network is partitioned. A simple connected (1-connected)

network might already be partitioned when a single Byzantine process exists, while a fully connected network (n-connected) can sustain more Byzantine nodes. Pease et al. proved that there exists a tight upper bound for f in these networks, namely $f < \lfloor N/3 \rfloor$ [14].

When a network is partially connected, Dolev showed that processes can still communicate reliably in the presence of f Byzantine nodes when the network is at least $2f + 1$-connected [1]. Dolev introduced two variants, where one has access to a routing table and one does not, the known and unknown topology variants respectively. For the unknown topology variant, the message is essentially flooded over the network, therefore following at least $2f + 1$ vertex-disjoint paths. In the routed variant messages are only transmitted over their predesignated routes. Since authenticated links[1] are assumed in this solution, every process can append the transmitter of a message to a header representing the message path. A process delivers a message when it has received the same payload data over $f + 1$ vertex-disjoint paths. Note that this means a Byzantine sender can cause only a single correct process to deliver a message, violating the basic principles of a Reliable Broadcast, which is why the broadcaster is assumed to be a correct process. We will only focus on the known topology variant.

Bracha described the *authenticated double echo* protocol [2] for fully connected networks, which gives the additional guarantee that either every correct process will deliver a message or none will, even when the broadcaster is Byzantine. This protocol uses three phases to coordinate the global acceptance of a message m: *send*, *echo*, and *ready*.

In their original versions, both protocols are less than practical. In the case of Dolev, the worst-case message and computational complexity are high ($\mathcal{O}(n!)$ for n processes), making it impractical for large ($n = 100$) networks. While Bracha is computationally less expensive, it requires a fully connected network, reducing its applicability in regular networks.

Bonomi et al. [5] introduced several improvements to Dolev's original protocol, considerably improving its average message complexity. These modifications make Dolev more practical for use in general networks, even though the complexity of delivery verification is still high.

Wang and Wattenhofer [3] introduced a combination of existing protocols, Bracha and Dolev for example, to use a protocol designed for a fully connected network (e.g. Bracha's protocol) on a k-connected (where $k < |V|$) network. More recently, Bonomi et al. [4] introduced several novel improvements to this protocol and combined it with an optimized version of Dolev's RC protocol [5; 6]. Their work showed significant improvements to the message complexity, and several modifications may also apply to other combinations, such as CPA [15] and Bracha.

2 System model and problem statement

Our model is defined by a set $\Delta = \{p_1, p_2, ..., p_N\}$ of N processes, uniquely identified by an identifier i known to all others. In the Byzantine fault model, it is assumed that there are at most $f < \lfloor N/3 \rfloor$ Byzantine nodes, which can exhibit arbitrary behavior.

Furthermore, the processes are connected by a network which can be represented by an undirected graph $G = (V, E)$. In this graph every vertex represents a process p_i, such that $p_i \in \Delta$, which means $V = \Delta$. The edges represent the communication links between nodes. Processes $p_i, p_j \in \Delta$ have a direct communication link if there exists an edge $(v_i, v_j) \in E$, which they can use to directly communicate with each other. If there exists no such link, they will have to rely on other processes to relay their messages. We assume that these links are authenticated, i.e., messages delivered at $p_i \in \Delta$ via edge $(v_i, v_j) \in E$ are guaranteed to originate from p_j, and vice-versa. In addition to this, the links are reliable, i.e., messages will always arrive at p_i if and only if p_j sent them over edge (v_i, v_j). However, there is no delivery time or delivery order guarantee, so a link can be synchronous or asynchronous. Graph G is known to all processes, and so are the identifiers for every process. Furthermore, it is assumed the network is static, i.e., the network topology does not change, and one or more processes can broadcast messages simultaneously. The processes are used as the underlying layer for application code, which receives data from the process when it delivers a message to the application layer.

To send message data to others, processes can add information to the message header, which can be used to uniquely identify the message and add protocol-specific information.

A Byzantine Reliable Broadcast (BRB) protocol guarantees the following properties:
(i) **Validity**: If process $p_i \in \Delta$ broadcasts message m, then every correct process $p_j \in \Delta$ delivers m at some point.
(ii) **No duplication**: A message m broadcast by process $p_i \in \Delta$ is not delivered more than once by every correct process $p_j \in \Delta$.
(iii) **Integrity**: If process $p_j \in \Delta$ delivers message m with sender p_i, process p_i has broadcast m in the past.
(iv) **Agreement**: If process $p_i \in \Delta$ delivers message m, then m will eventually be delivered by every correct process $p_j \in \Delta$.

We will be introducing improvements to both Dolev [1], Bracha [2], and Bracha-Dolev [3], which make different assumptions about the network G and provide different guaranteed properties.

Dolev assumes a network G that is at least $2f + 1$-connected. Furthermore, Dolev provides Reliable Communication (RC) which guarantees the same properties as BRB, except for the **Agreement** property. Bracha assumes a fully connected network G, i.e. for every pair $v_i, v_j \in V$ there exists an edge $(v_i, v_j) \in E$. Unlike Dolev, Bracha guarantees all BRB properties.

We make the same assumptions as the mentioned protocols while adding topology knowledge and static networks.

Reducing the number of messages

While all mentioned protocols work well in their designed environments, there is naturally a substantial amount of redundant work when processes are unaware of the network topology.

This paper aims to reduce the number of messages which are transmitted through the network for the three mentioned protocols. This reduces the network usage even further, assuming that all processes know the network topology. In addition to this, it might be possible to improve the delivery complexity of Dolev in the process by taking advantage of the fact that messages will traverse fixed paths. Furthermore, while the general process for handling known topologies for Dolev has been described in the original paper [1], no actual implementation has been provided, which is something this paper will also do and is an additional contribution of this work.

3 Background

In this section, we will explain Dolev's and Bracha's protocols, and how they can be combined into Bracha-Dolev.

[1] Authenticated links guarantee messages sent over a link originate from the complementing process

Dolev

Dolev's protocol provides reliable communication when the network has authenticated links and is at least $2f + 1$-connected.

When a message traverses the network, processes leverage the authenticated links to build a traversal path for each message. Said paths have two purposes: avoiding transmission loops and message verification. The former is at play when processes relay messages to their neighbors; a message is forwarded to all neighbors, except to the transmitter and processes which are already present in the path. This is required to avoid messages circulating through the network indefinitely. The paths are also used for verification; a message is delivered whenever it has been received over $f + 1$ disjoint paths.

The basis for the correctness for Dolev's protocol lies in Menger's theorem [16] which shows that there exist $2f + 1$ disjoint paths between every pair of processes if a network is $2f + 1$-connected, and the fact that at most f of those paths can contain one or more Byzantine processes.

Pseudocode for Dolev's protocol for a single message is provided in Algorithm 2 in Appendix A.1.

Bracha

Unlike Dolev's protocol, Bracha's protocol requires a fully connected network while guaranteeing all four BRB properties, including the **Agreement** property. The protocol has three phases: *send*, *echo*, and *ready*.

When a process wants to broadcast a message it sends the payload in a *send* message to all processes, including itself. When a process receives a *send* messages, it responds by transmitting an *echo* message to all processes with the corresponding content. Every process then waits for a minimum of $\lceil \frac{N+f+1}{2} \rceil$ *echo* messages. After this number has been reached, or $f + 1$ *ready* messages have been received, a process will transmit its own *ready* message to all processes. Finally, a message will be delivered when $2f + 1$ corresponding *ready* messages have been received, as can be seen for a single message in Algorithm 3 in Appendix A.1.

Bracha-Dolev

Dolev's and Bracha's protocol can be combined to achieve BRB guarantees in a multi-hop network, as described in [3].

This works by layering the two protocols, where Dolev's protocol forms the bottom layer. This means that every Bracha broadcast operation is replaced by a Dolev broadcast, and every Bracha receive operation by a Dolev deliver.

By layering Bracha and Dolev, the latter emulates a fully connected network by enabling the former to reliably reach all processes. However, this means the message complexity of both protocols is essentially multiplied.

Instead of simply layering the two protocols, a cross-layer version [4] can be used which allows for greater optimization.

4 Improving Dolev on known topologies

In this section, we will describe the algorithms required to leverage the potential of topology knowledge, in what ways Dolev's protocol will have to be modified for this case, and **7** modifications to the resulting protocol.

4.1 Finding k-disjoint paths

To build a routing table, one has to find k vertex-disjoint paths to every $p_i \in \Delta$ where the total weight of all paths is minimized. Formally this problem is known as the *min-sum disjoint paths problem*.

A straightforward solution would be to repeatedly find the shortest path, remove the edges in the path, and repeat this process k times. However, even though this algorithm would work on most graphs, there exist so-called *trap topologies* for which this algorithm would fail to find a solution. In said topologies there exists a path with a minimal sum, which traverses multiple disjoint paths, effectively blocking off more disjoint paths than needed. An example of a trap topology can be found in Figure 1. In this example the path *a-c-b-d* would be chosen over *a-b-d* and *a-c-d* by this naive algorithm.

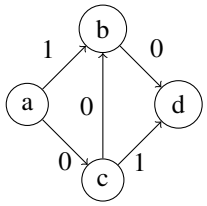

Figure 1: In this example there exist two disjoint paths from a to d, but only one would be found by a naive shortest path algorithm

A solution that can handle trap topologies was introduced by Bhandari [17]. This algorithm finds k edge-disjoint paths in a directed weighted graph by repeatedly finding the shortest path and inverting the resulting edges. An edge is inverted by simply reversing its direction and multiplying its weight by -1. If there already exists a reverse edge for the edge being inverted, the existing edge is replaced. If the edge that is being inverted has already been inverted once, it can be simply discarded instead. To find the result, all complementing edges are removed from the set with all edges in the paths. The final paths can then easily be retrieved from the resulting sets, as every edge will only have two or fewer matching edges.

Note that this algorithm only returns k edge-disjoint paths, not k vertex-disjoint paths. This problem can be solved by applying a process called *vertex splitting*, which as the name implies splits every vertex except for the source and sink into two distinct vertices. A vertex is split into an 'in' vertex, and an 'out' vertex. Every incoming edge will be directed to the former, while every outgoing edge will be directed to the latter. The two vertices are connected by a directed edge with a weight of zero from the 'in' vertex to the 'out' vertex. This process is visualized in Figure 2. Note that this change forces every path which uses a vertex to use the interconnecting edge, limiting the number of times every vertex can be used to one. This means the algorithm will now find k vertex-disjoint paths.

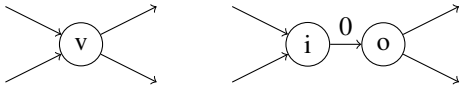

Figure 2: Vertex splitting visualized

To build the full routing table, this algorithm has to be completed for every process, resulting in $(n - 1) \times (2f + 1)$ paths which will reach every process over $2f + 1$ node-disjoint paths.

The pseudocode for the k-disjoint path solver can be found in Algorithm 1. We use the Shortest Path Faster Algorithm or SPFA [18; 19], which is a queue-based Bellman-Ford [20; 21] variation to find the shortest path in our paper, but every algorithm that is capable of handling negative weights can be used.

A single entry in our routing table can be created using our disjoint path solver, which uses Bhandari's [17] algorithm and

SPFA [18; 19] to find the disjoint paths with a minimum sum of weights. The full table can be created by varying the target process t, by iterating over all possible values for t.

Algorithm 1: Disjoint path solver algorithm

```
1  func DisjointPaths(g, s, t, k):
2      edges = DisjointEdges(g, s, t, k)
3      filtered = FilterCounterparts(edges)
4      return BuildPaths(filtered, s, t, k)
5  func DisjointEdges(g, s, t, k):
6      split = VertexSplitting(g)
7      result = ∅
8      repeat k times
9          path = ShortestPath(s, t, split)
10         forall e ∈ path do
11             result.add(e)
12             InverseEdge(split, e)
13     return result
14 func FilterCounterparts(edges):
15     drop = result = ∅
16     forall (f, t) ∈ edges do
17         drop.add((t, f))
18     forall e ∈ edges do
19         if not drop.contains(e) then
20             result.add(e)
21     return result
22 func BuildPaths(edges, s, t):
23     result = ∅
24     forall (f, e) ∈ edges do
25         if f = s then
26             path = ∅
27             while not e = t do
28                 path.add((f, e))
29                 (f, e) = Next(e)
30             path.add((f, e))
31             result.add(path)
32     return result
```

4.2 Modifying Dolev

We can distinguish between two options for the routing table in a modified version of Dolev's protocol.

In one version a process only computes its own routing table. This is computationally less expensive but requires more information to be included in the transmitted messages, as other processes are unaware of the desires paths of messages. Message verification is slightly less complex than in the case of normal Dolev since traversed paths can be remembered by receiving nodes. However, the first message will have to be verified using the same technique as in an unknown topology.

In the second version, every process computes the routing table for every other process. This is computationally expensive but reduces the amount of information in the message headers considerably. Message verification also becomes trivial, as every process is aware of the paths the messages will use, so any message with an incorrect path can be discarded. Care has to be taken that this process is deterministic, as to avoid having different routing tables for different processes.

Note that the computational cost is only a one-time cost with the assumptions we use; static topologies. When dynamic topologies are used, the computational cost becomes more important.

In this paper, we have opted for the protocol where every process has access to every routing table to decrease the message size, as will be discussed later.

Verification

The message verification is simplified greatly, as every message path can now be simply compared to the corresponding routing table entry. If no matching entry exists for the given origin, the message is discarded. Otherwise, it is kept in memory. Once enough messages with identical payloads and unique paths have been received the message is delivered. This can easily be achieved by creating a mapping between a message identifier, consisting of the regular Dolev identifier and the hash of the payload, and a set of paths. When the size of the set of paths is equal to $f + 1$, corresponding content can be delivered.

4.3 Optimizations on routing table

In addition to providing a base implementation for Dolev's protocol with routing, we also introduce several optimizations to further reduce the number of messages transmitted. To avoid confusion we use the identifier **ORD.1-7** for our optimizations. This section will elaborate on optimizations focused on reducing the size of the routing table, decreasing the memory usage in the process.

ORD.1: Avoid transmitting subpaths

When process p is building its routing table, it can discard all paths which are a subpath of other paths. The messages related to said paths can be dropped without loss of information, as it is guaranteed another message will traverse the path in full. This optimization will reduce the size of the routing table which reduces the number of messages transmitted and the memory usage of the routing table, or -if combined with **ORD.3**- reduce the amount of information being transmitted with the message. An example is illustrated in Figure 3.

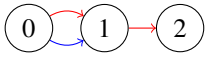

Figure 3: The blue path can be dropped, as it is a full subpath of the red path

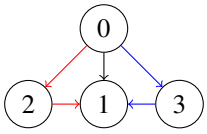

Figure 4: Both blue and red paths can be dropped, as only one direct path to a neighbor is required

ORD.2: Use a single route for direct neighbors

Bonomi et al. [5] showed that direct neighbors can directly deliver messages originating from the source. A similar change can be made to the routed version of Dolev's protocol, by accepting only one path to direct neighbors. We have achieved this by adding links to neighboring processes separately before finding disjoint edges, which corresponds to line 2 in Algorithm 1. An example is given in Figure 4.

ORD.3: Merge next hops when broadcasting

When process p is transmitting the initial broadcast messages, it can merge all messages which have the same first hop into a single message. After a process receives these merged messages, the original messages can be reconstructed. The process can then be continued by all relaying nodes until only a single base message remains. This means the desired and traversed path form a pair which needs to be maintained throughout the entire

network. This optimization applies to the creation of routing tables but is also applied when processes disseminate messages as they may need to split messages. This process is shown in Figure 5.

Remark The message header will contain multiple paths after this change, for which the actual paths will be identical until the messages are split. It might be possible to further reduce the header size by only including a single copy of the actual path when there exist multiple identical copies.

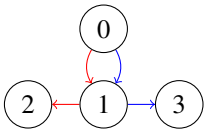

Figure 5: The messages can be merged to traverse (0,1) together, and then split at 1

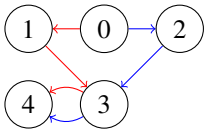

Figure 6: The messages can be merged at 3 to traverse (3,4) together

ORD.4: Reuse paths when possible

When messages traverse the same path, processes can attempt to merge messages as explained in **ORD.3**. For this reason, routes should be as similar as possible. We have achieved this by adding weights to unused edges after each iteration of the disjoint k-paths solver, which corresponds to the space between lines 12 and 13 in Algorithm 1.

Additional care has to be taken when **ORD.2** is also applied, to avoid routing messages to neighbors over intermediate nodes.

4.4 General optimizations

This section solely focuses on optimizations that do not apply to the creation of the routing table or are mostly applicable at message dissemination.

ORD.5: Apply delayed relaying and merging

While **ORD.3** introduced the concept of merging messages, this is a *structurally decreasing* process, i.e. the number of wrapped messages in a single message will only decrease as the message is being relayed. The reason for this is that processes only analyze an incoming message without additional context, which means a process will inspect the incoming buffer sequentially and immediately relay messages when possible. While this is pure[2] there are cases when using the context of other messages or delaying outbound messages is beneficial. For example, two messages with the same Dolev identifier received over two different links can be merged into a single message (similar to **ORD.3**) when they share the same next-hop. However, since these messages are handled separately the process needs to delay the former and use its context when processing the latter message. This situation has been illustrated in Figure 6.

One possible option is to only relay messages whose contents have been delivered and keep other messages in a buffer that can be used to merge outbound messages. While this approach would work on some networks, a deadlock will occur when processes are delaying messages which would otherwise cause the other process to deliver.

This can be avoided by detecting possible deadlocks and then marking one of the conflicting paths as a priority path, which means processes will immediately relay it. Deadlocks can be detected by finding a pair of paths for which at least one edge whose reverse edge is contained in the other path exists. Deciding on priority paths can be done in any way, but at least one path must be picked for every conflict. In this paper, we simply find the processes in an overlapping section with a maximum distance and mark the path which traverses the process with the smallest ID first as a priority path, unless the other path is already marked as a priority path. An optimal solution to this problem exists, but this is outside the scope of this paper.

Remark This modification introduces more latency to the protocol, as (some) messages are being held in buffers for longer amounts of time. This can be partially mitigated by applying optimizations to the deciding procedure. For example, designating paths as a priority path when all processes on the conflicting edges only have to relay that single message, since there is no other message to merge with. Another addition might be *piggybacking*, which means messages in the buffer can be merged with a priority message sharing the same next-hop since the priority message will be transmitted anyways.

ORD.6: Merging messages with identical contents

While most optimizations focus on single-message broadcasts, i.e., there is only one process broadcasting a message, there exist plenty of algorithms where every process transmits messages simultaneously. For the general case, there is not a lot that one can optimize for multiple broadcasts. However, in the case where the payloads of multiple messages are identical, there is room for improvement. Examples of these cases are *keepalive* or topology discovery protocols, where the payload will likely be identical for all processes.

When the payload is identical processes can combine multiple messages into a single wrapper message, reducing the amount of time the payload is transmitted. This modification can use the buffer created by **ORD.5**. By tracking the Dolev identifiers for identical payloads, the buffers of multiple messages can be queried when relaying messages.

A receiving process can reconstruct the original Dolev messages based on a single wrapper message, reducing the amount of information transmitted in these messages.

Remark Something similar might be possible for Bracha, by tracking similar echo payloads. Whenever multiple similar payloads exist a Bracha process can wait to send the appropriate *readys* until all similar payloads have enough echo confirmations. There should be an early exit strategy to avoid waiting indefinitely, but this will need to be researched further.

ORD.7: Implicit desired paths

As discussed in Sec. 4.2, the routing information can be included in the message headers to reduce computational complexity or it can be fully precomputed to reduce the message size. When optimizing for bandwidth usage the latter is the preferred option.

This modification ensures message headers are not larger than needed by precomputing the routing tables for every process, which can then be used to deduce the desired paths based on the actual paths. Based on the actual paths the matching desired paths are retrieved from the global routing table. Depending on the other active modification, one or all of the desired paths are used to relay the message.

Remark If this modification is used in the context of topology discovery, it might be possible to reduce the size of the routing table while it is being transmitted over the network. Some entries are no longer applicable and can then be dropped from the broadcast. However, this is for future research, as this paper does not focus on integration with topology discovery already.

[2]Pure in the functional programming sense, a message enters the pipeline and zero or more come out without using other context

5 Bracha on known topologies

In the case of Bracha's protocol, topology knowledge is not as useful as with Dolev's protocol. This is because Bracha assumes a fully connected network, which means the topology is known anyways. The only knowledge processes gain is the weight of edges representing links between other processes. We will try to use this knowledge for our optimizations

Similar to Dolev's optimizations, we will use **ORB.1-2** to identify different optimizations.

ORB.1: Implicit echo messages

Instead of sending a *send* message and an *echo* message separately, a process can send a single *send* message and others will interpret that as a combined *send* and *echo* message. Similarly, an *echo* or *ready* message will also be interpreted as a *send* message. This optimization is similar to **MDB.2** from [4], as that optimization converts the *send* message into an *echo* message after the first hop. While this is slightly different, the effects are nearly identical.

ORB.2: Use minimal subset of neighbours

Bracha's protocol requires $\lceil \frac{N+f+1}{2} \rceil + f$ participants in the *echo* phase and $3f + 1$ for the *ready* phase. This means that for over-provisioned networks, i.e., networks where $f < \lfloor \frac{N}{3} \rfloor - 1$, we can avoid using all processes in said phases.

This is similar to the optimization **MBD.11** from [4]. However, we can improve overall latency by assigning a cost to every neighbor based on their outgoing edges and then making a selection.

There are several ways to assign a cost to a process. Simple heuristics include finding the minimum sum of weights of edges used, finding the minimum sum of weights for all edges, and several other similar approaches. The optimal solution can also be computed, but that is outside of our scope. In this paper, we use the simple heuristic of finding the minimum sum of weights for all edges.

Using the chosen heuristic, every process calculates a Bracha routing table which contains all *echo* and *ready* participants for every message origin.

To not add information to the message header, we made every process precompute these participant tables. Processes can then use these tables to find the participant sets for a given origin.

6 Bracha-Dolev on known topologies

In this section, we will describe how our previous optimizations for Dolev and Bracha can be applied to Bracha-Dolev and additional cross-layer optimizations.

6.1 Applying optimizations

As Dolev is used as the lowest layer, all **ORD** optimizations can be applied as-is to our improved version of Bracha-Dolev.

Bracha is used as the upper layer in Bracha-Dolev, and as such we can not directly apply **ORB.2**, since it assumes a fully connected network. However, we can use a different way of selecting processes, by simply selecting the closest processes in the network. The other Bracha optimization, **ORB.1**, can be directly applied as it does not rely on topology knowledge.

6.2 Optimizations

In addition to applying all **ORD** and (modified) **ORB** optimizations, we can also apply some new modifications. These are identified by **ORBD.1-2**.

ORBD.1: Using partial Dolev broadcast

When **ORB.2** is active not all processes are participating in the *echo* phase, and therefore do not need to receive these messages. However, by default Bracha-Dolev can only perform full broadcasts on the Dolev layer. This optimization changes that by allowing partial broadcasts on the Dolev layer, i.e., some messages are not delivered by all processes. While this violates the RC properties for Dolev, the overall Bracha-Dolev guarantees still hold, so this is valid.

We have added this modification by adding an additional precomputed routing table, which takes the Bracha phase and message origin into account. Dolev will now inspect the Bracha message type (*send*, *echo*, or *ready*) to determine which routing table to use, and transmit the messages accordingly.

ORBD.2: Merging multiple Bracha messages

The Dolev layer considers messages originating from different processes as different Dolev broadcasts altogether, which is technically correct even though they may all originate from the same Bracha broadcast. However, Bracha messages from the same Bracha-Dolev broadcast share identical payload and origin data. This can be leveraged on the Dolev layer by identifying Bracha messages belonging to the same Bracha broadcast and merging them if possible, by utilizing the buffer created by **ORD.5**.

When merging messages, a special wrapper message is transmitted by a Dolev node, which neighbors can use to reconstruct the original messages, similar to the wrapper message in **ORD.6**. This wrapper message includes the original payload and origin data, and the regular Dolev data for all merged messages. This reduces the number of times the payload is transmitted, at the cost of some additional header information.

Remark This optimization can likely be extended to the Bracha layer in addition to being just on the Dolev layer, to leverage topology knowledge even more. However, at this time we have no solution to this problem, but also no proof of its infeasibility. This should be further explored in the future.

7 Evaluation

In this section, we will discuss the methodology we used and the results of our optimizations.

7.1 Methodology

For our research we implemented an evaluation program in Go which uses goroutines [22] as a process abstraction, and dedicated channels [23] as communication links. The protocol instances are instantiated by the process wrappers, and they have access to a network and an application instance, which are defined by the interfaces containing `Send(dst, m)` and `Deliver(m)` respectively. The protocols themselves provide the `Init()`, `Receive(src, m)`, and `Broadcast(m)` functions.

In addition to the original protocols and improved version of Dolev [5] and a version of Dolev with naive routing was implemented. These two versions are the baseline for Dolev and Bracha-Dolev.

We focus on message complexity and network consumption, which is defined by the total number of messages transmitted and the total number of bytes transmitted, respectively. We mention latency when notable, but this is not a statistic we focus on. We define latency as the time between the original broadcast and the final non-Byzantine node delivering the message.

We use similar graphs as used in [4; 5]: generalized wheels, multipartite wheels, and random regular graphs. For the tests,

we use an AMD Ryzen 5 2600 (3.4-3.9GHz) machine. The usage of channels leads to a different throughput per machine, but their performance will not limit the tests and will not affect our main measurement.

We will run the tests with varying random graphs, broadcasting process, byzantine processes, and parameters N, k, f, such that $N \geq 3f + 1$ and $k \geq 2f + 1$, and report the mean and standard deviation of five tests. In most tests, a single process will broadcast a single message, unless the modifications being tested include **ORD.6** as it is specifically made for the case of multiple broadcasters. In that case, the amount of transmitters m is defined by $N - f$.

Remark Note that the latency will not be entirely representative of the latency in a real deployment, as our simulated links have low latency which means latency is largely influenced by computing time.

7.2 Impact of individual optimizations

We evaluated the impact of individual optimizations on message complexity and network consumption. Table 1 summarizes our findings for every individual modification compared to its baseline. The baseline is different for each protocol: for Dolev, we compare to a version with naive routing, for Bracha we compare to the original version, and for Bracha-Dolev we compare to a version of Bracha-Dolev which uses naive routing for the Dolev layer and the original Bracha implementation. For these tests, random graphs were used with a size of $N = 150$ for Dolev and Bracha and $N = 75$ for Bracha-Dolev, and we varied the k and f to find out when modifications are useful.

We will illustrate some modifications with the aforementioned configuration.

There are several modification which perform well across the board, such as **ORD.1-3,7**, **ORB.1,2**, and **ORBD.1**. Others are slightly more nuanced, however. For example, both **ORD.6** and **ORBD.2** perform better when the payload is large since they both rely on merging the payload while adding slightly more information in a single header. The opposite is true for **ORD.7**, which performs better when there is a smaller payload. This is because this optimization tries to minimize the information contained in the message header, which is insignificant if a large part of the message consists of the payload data.

Another optimization, **ORD.5**, does not show significant improvement. However, this lack of performance is offset by the fact that both **ORD.6** and **ORBD.2** rely on this modification. Another modification not showing significant improvements is **ORD.4**. While a part of this is likely caused by a non-optimal algorithm to reuse paths, it can also be attributed to the fact that it heavily relies on **ORD.3** to complete its task and needs an oversaturated network.

It is also interesting to note the dependencies between modifications. For example, **ORDB.2** on its own does not improve the protocol that much. However, when combined with **ORD.2** and **ORD.3** the number of messages merged increases more than tenfold. The reason being that these two modifications cause a lot of messages to end up in the buffer and also cause quicker deliveries, leading to more merging in **ORDB.2**.

The opposite is also true, some modifications are mutually exclusive. For example, **ORD.6** is unable to merge messages when **ORBD.2** is active since they share the same buffer and **ORBD.2** changes the payload temporarily. For this reason, it is recommended to prefer **ORD.6** over **ORBD.2** when all processes are broadcasting identical payloads.

7.3 Improvements

In addition to comparing individual modifications, we will evaluate the performance of our fully modified protocols. Figure 7 shows the reduction of our protocol compared to Dolev, Bracha, and Bracha-Dolev with regards to the message complexity. The reduction is relative to the same baseline used for the individual modifications.

For Dolev and Bracha-Dolev we again use random graphs with $N = 150$ and $N = 75$ respectively, and vary the connectivity k. The number of Byzantine nodes f is defined by $\lfloor \frac{k-1}{2} \rfloor$. In the case of Bracha we have can only use fully connected graphs, and will therefore vary the number of processes N depending on the connectivity. The number of byzantine nodes, in this case, is defined by $\lfloor \frac{k}{4} \rfloor$. In all cases, the payload size is equal to 12B.

These tests show we can achieve a mean reduction of 79.49% (+/-0.93%) for Dolev, 23.32% for Bracha, and 89.54% (+/-0.22%) for Bracha-Dolev under the conditions mentioned above. The reduction in bytes transmitted is similar: 85.86% (+/-0.68%), 23.32%, and 92.32% (+/-0.17%) respectively.

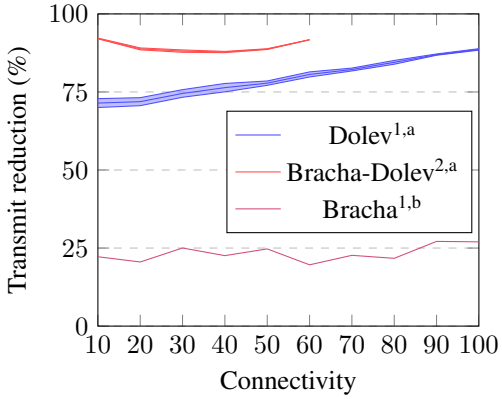

Figure 7: Reduction of message complexity using K-random graphs and fully-connected graphs (Bracha), while varying the connectivity. [1] $N = 150$, [2] $N = 75$, [a] $f = \lfloor \frac{k-1}{2} \rfloor$, [b] $f = \lfloor \frac{k}{4} \rfloor$

7.4 Scalability

In real deployments, the number of processes in the network will likely scale quickly, which is why we also evaluated the scalability of the protocol for an increasing number of processes. We considered graphs that include 25 to 150 processes in increments of 25. The connectivity k and Byzantine parameter f are defined as $k = \lfloor \frac{N}{3} \rfloor$ and $f = \lfloor \frac{k-1}{2} \rfloor$ for (Bracha-)Dolev and $f = \lfloor \frac{k}{4} \rfloor$ for Bracha. The other configuration is identical to the previous sections.

The evaluation for Bracha-Dolev was unable to continue after 75 processes, due to resource constraints; the version with naive routing and no additional optimizations was using too much memory[3] during testing. We expect the trend of outperforming the base version on larger networks to continue, leading us to believe the reduction would be around 87% for the larger networks.

From these experiments we can see that the message complexity reduction is not decreasing, which means in terms of message complexity and network usage our modified versions scale well with the number of processes. However, the latency is still doubling after each increment, suggesting exponential growth. The modified protocols still outperformed the baseline

[3] 12GiB on a 16GiB system in this case

	Small payload (12B)				Large payload (12KB)			
ID	**Msg. red. %**	**Useful when**	**Usage red. %**	**Useful when**	**Msg. red. %**	**Useful when**	**Usage red. %**	**Useful when**
ORD.1	10.18% (+/-2.65%)	small $k\wedge$ large $f*$	8.29% (+/-3.05%)	small $k\wedge$ large $f*$	8.67% (+/-3.47%)	small $k\wedge$ large $f*$	8.63% (+/-3.48%)	small $k\wedge$ large $f*$
ORD.2	34.65% (+/-2.41%)	large $k*$	34.82% (+/-2.78%)	large $k*$	32.71% (+/-3.01%)	large $k*$	32.70% (+/-3.01%)	large $k*$
ORD.3	63.06% (+/-1.37%)	always	10.11% (+/-2.94%)	always	62.03% (+/-1.77%)	always	61.29% (+/-1.80%)	always
ORD.4	0.84% (+/-3.04%)	large f	0.77% (+/-3.46%)	large f	-0.90% (+/-3.81%)	-	-0.90% (+/-3.82%)	-
ORD.5	2.09% (+/-2.99%)	always	1.63% (+/-3.43%)	always	-0.06% (+/-4.04%)	-	0.07% (4.05%)	-
ORD.6	6.18% (+/-0.33%)	small $f*$	0.81% (+/-0.22%)	small f	6.41% (+/-0.19%)	small f	6.31% (+/-0.18%)	small f
ORD.7	1.41% (+/-3.09%)	-	66.15% (+/-1.38%)	always	0.05% (+/-4.51%)	-	0.81% (+/-4.47%)	-
ORB.1	0.41% (+/-0%)	always	0.41% (+/-0%)	always	0.41% (+/-0%)	always	0.41% (+/-0%)	always
ORB.2	41.73% (+/-0%)	small f	41.73% (+/-0%)	small f	41.73% (+/-0%)	small f	40.91% (+/-0%)	small f
ORBD.1	24.51% (+/-2.71%)	small $k\wedge$ small $f*$	24.68% (+/-2.73%)	small $k\wedge$ small $f*$	21.66% (+/-2.03%)	small $k\wedge$ small $f*$	21.66% (+/-2.03%)	small $k\wedge$ small $f*$
ORBD.2	1.25% (+/-1.12%)	always	-3.04% (+/-1.96%)	never	0.2% (+/-0.82%)	-	0.14% (+/-0.83%)	-

Table 1: Effect of modifications measured on random graphs compared to their respective protocol standard. The mean reduction and standard error are listed respectively, in addition to a small description of the best use-cases. Note that descriptions marked with a star* are always useful, but will perform best in the given use-case.

in terms of latency by 25.16%, 22.38%, and 50.19% for Dolev, Bracha, and Bracha-Dolev respectively.

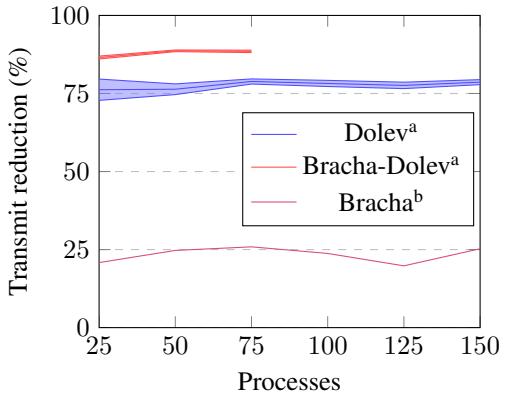

Figure 8: Reduction of message complexity using K-random graphs and fully-connected graphs (Bracha), while varying the number of processes. $k = \lfloor \frac{N}{3} \rfloor$, [a] $f = \lfloor \frac{k-1}{2} \rfloor$, [b] $f = \lfloor \frac{k}{4} \rfloor$

7.5 Discussion

While our results are promising, we have focused on two main statistics: message complexity and network usage. This means that other statistics such as latency have sometimes been sacrificed to enhance our chosen statistics, as is the case with **ORD.6** and **ORBD.2** for example. This might not be desired in some systems.

As mentioned earlier, the measured latency is not fully representative of the real world. Something similar is true for the measured network usage, as we use the size of internal structures as measurement. While this size is mostly representative of the actual size, it also includes some internal headers which would not be transmitted, and should therefore not be included. However, this will have no significant impact on our results as all measurements will include a similar size for internal data, which means the relative reductions will not be affected.

Our evaluation was completed on a simulated network using Go channels [23], which limits our evaluation to a single machine with no bandwidth limits. A more elaborate evaluation should use multiple systems, to evaluate the protocol on a real network. Our evaluation platform can be relatively easily modified to use a framework such as ZeroMQ [24] or plain TCP sockets, as the network evaluation layer is completely abstracted from the protocol layer. However, our evaluation platform also uses a controller to run processes and set correct parameters, which should also be modified to work on multiple machines. For the sake of time, this was not implemented for this paper, but the proper abstraction was used to have the possibility for extension for future papers.

Some optimizations introduced by Bonomi et al. [4] can also be translated to our protocol, which could reduce the message complexity even further. Some have already been used (**MBD.2**, **MBD.12**), while others could be implemented with (**MBD.3**, **MBD.4**) or without (**MBD.7**, **MBD.8**) slight modifications.

We can safely conclude that we can indeed reduce the number of messages when leveraging topology knowledge, but the system model might be too strict for modern networks as they are generally dynamic instead of static.

8 Broader impact and reproducibility

Our research focused on improving existing protocols. This means we guarantee the same properties as the original protocols while putting the network under less stress in certain systems. For this reason, there are no inherent risks to our work. In addition to this, there are limited malicious uses for our work, as it works as an underlying protocol similar to the regular internet protocol. However, our modifications add a considerable amount of complexity to the protocol, allowing for more developer error possibly leading to violated protocol properties.

Now that we have discussed the broader impact of our work, its reproducibility should also be mentioned. All of our results are retrieved from a standalone binary whose source will be published together with this paper. The program has a low barrier of entry and can be used by anyone since the program is written in a widely supported language (Go) and has no other system dependencies. The program can be found in the GitLab repository[4].

The exact configurations used can be deduced from the evaluation section, and are also included in the provided command-line tool. These can then be executed by the program. The exact graphs used for evaluation will also be published along with the code, although a user can also choose to generate new graphs.

It is important to note that results may be different for every computer, as the program will execute everything as fast as possible. However, the relative differences between the original protocols and our improved versions will be closer to the results showed in our paper.

9 Conclusion and Future work

In this paper, we have introduced the Byzantine Reliable Broadcast problem on partially connected networks and fully connected networks where the topology is known to all processes.

[4] https://gitlab.tudelft.nl/jdecouchant/rp21-group31-4-anema

We started by introducing the current state of the problem and how the original protocols work. We continued by elaborating on how one can find the required paths through the network for Dolev, and then how this information can be used to build routing tables for processes. We then described **9** modifications to Dolev, Bracha, and Bracha-Dolev, and evaluated each modification separately. When we combine all modifications, we provide a solution with a lower message complexity and network usage than existing solutions, a reduction of 71.9% and 79.4% respectively with a 12B payload when $N = 150$ and $f = 20$ for Dolev. We have concluded that we can indeed reduce the number of messages transmitted when processes have topology knowledge.

This work can be extended in the future by deploying our modified protocols on real infrastructure to get accurate measurements as opposed to simulations. Furthermore, the disjoint path solver can likely be further optimized by enhancing the pathfinding and (re)using better-suited data structures. Applying our modifications to dynamic networks should also be researched further. In addition to this, we use several simple heuristics in our paper (**ORB.2**, **ORD.5**) which should be replaced by optimal solutions or improved heuristics. Another interesting direction is that of topology discovery [10; 11; 12]. Previously unknown networks can use optimizations specific for known networks when matched with a topology discovered. One challenge to overcome in this scenario is the tolerance for imperfect routing tables.

References

[1] D. Dolev, "Unanimity in an unknown and unreliable environment," in *FOCS*, 1981.

[2] "Asynchronous byzantine agreement protocols," *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987.

[3] Y. Wang and R. Wattenhofer, "Asynchronous byzantine agreement in incomplete networks." ACM, 2020.

[4] S. Bonomi, J. Decouchant, G. Farina, V. Rahli, and S. Tixeuil, "Practical byzantine reliable broadcast on partially connected networks," 2021.

[5] S. Bonomi, G. Farina, and S. Tixeuil, "Multi-hop byzantine reliable broadcast with honest dealer made practical," 2019.

[6] "An efficient algorithm for byzantine agreement without authentication," *Information and Control*, vol. 52, no. 3, pp. 257–274, 1982.

[7] D. Dolev and R. Reischuk, "Bounds on information exchange for byzantine agreement," vol. 32, no. 1, 1985.

[8] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren, "Synchronous byzantine agreement with expected o(1) rounds, expected $o(n^2)$ communication, and optimal resilience," in *Financial Cryptography and Data Security*, 2019.

[9] L. Tseng, Y. Wu, H. Pan, M. Aloqaily, and A. Boukerche, "Reliable broadcast in networks with trusted nodes," in *2019 IEEE GLOBECOM*, 2019, pp. 1–6.

[10] S. Dolev, O. Liba, and E. M. Schiller, "Self-stabilizing byzantine resilient topology discovery and message delivery," in *Networked Systems*, 2013.

[11] M. Nesterenko and S. Tixeuil, "Discovering network topology in the presence of byzantine faults," in *Structural Information and Communication Complexity*. Springer, 2006, pp. 212–226.

[12] G. Farina, "Tractable reliable communication in compromised networks," Ph.D. dissertation, 12 2020.

[13] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 203–226.

[14] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.

[15] "Broadcasting with locally bounded byzantine faults," *Information Processing Letters*, vol. 93, no. 3, pp. 109–115, 2005.

[16] K. Menger, "Zur allgemeinen kurventheorie," *Fundamenta Mathematicae*, vol. 10, no. 1, pp. 96–115, 1927.

[17] R. Bhandari, "Optimal physical diversity algorithms and survivable networks," in *Proceedings Second IEEE Symposium on Computer and Communications*, 1997, pp. 433–441.

[18] E. Moore, *The Shortest Path Through a Maze*, ser. Bell Telephone System. Technical publications. monograph.

[19] D. Fanding, "A faster algorithm for shortest-path–spfa," *Journal of Southwest Jiaotong University*, vol. 2, no. 9, pp. 207–212, 1994.

[20] R. Bellman, "On a routing problem," *Quarterly of applied mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[21] L. R. Ford Jr, "Network flow theory," Rand Corp Santa Monica Ca, Tech. Rep., 1956.

[22] "Goroutines," https://tour.golang.org/concurrency/1, accessed: 2021-06-12.

[23] "Channels in go," https://tour.golang.org/concurrency/21, accessed: 2021-06-12.

[24] "Zeromq," https://zeromq.org/, accessed: 2021-06-24.

A Appendix

A.1 Pseudocode

Algorithm 2: Dolev's Reliable Communication routed algorithm

```
1  On event Init:
2      delivered = False
3      paths = ∅
4  On event Receive(p_recv, m, path, planned):
5      path = path ∪ {p_recv}
6      forall p_j ∈ planned do
7          transmit(p_j, m, path, planned)
8      paths.add(path)
9      if paths contains f + 1 node-disjoint paths to the
         origin and delivered = False then
10         deliver(m)
11         delivered = True
12 On event Broadcast(m):
13     deliver(m)
14     delivered = True
15     forall (p_j, route) ∈ routingTable do
16         transmit(p_j, m, ∅, route)
```

Algorithm 3: Bracha's authenticated double echo algorithm

On event `Init`**:**
    sentEcho = sentReady = delivered = False
    echos = readys = $\emptyset$
On event `ReceiveEcho`(p_{recv}, m)**:**
    if ***not*** *sentEcho* **then**
        forall $p_j \in neighbours$ **do**
            transmit(p_j, m, ECHO)
        sentEcho = True
    echos.add(p_{recv})
    if *len(echos)* $\geq \lceil \frac{N+f+1}{2} \rceil$ ***and not*** *sentReady* **then**
        forall $p_j \in neighbours$ **do**
            transmit(p_j, m, READY)
        sentReady = True
On event `ReceiveReady`(p_{recv}, m)**:**
    readys.add(p_{recv})
    if *len(readys)* $\geq f + 1$ ***and not*** *sentReady* **then**
        forall $p_j \in neighbours$ **do**
            transmit(p_j, m, READY)
        sentReady = True
    if *len(readys)* $\geq 2f + 1$ ***and not*** *delivered* **then**
        deliver(m)
        delivered = True
On event `Broadcast`(m)**:**
    forall $p_j \in neighbours$ **do**
        transmit(p_j, m, SEND)
        transmit(p_j, m, ECHO)